**1:-A\* ALGORITHM**

def aStarAlgo(start\_node, stop\_node):

open\_set = set(start\_node)

closed\_set = set()

g = {}

parents = {}

g[start\_node] = 0

parents[start\_node] = start\_node

while len(open\_set) > 0:

n = None

for v in open\_set:

if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):

n = v

if n == stop\_node or Graph\_nodes[n] == None:

pass

else:

for (m, weight) in get\_neighbors(n):

if m not in open\_set and m not in closed\_set:

open\_set.add(m)

parents[m] = n

g[m] = g[n] + weight

else:

if g[m] > g[n] + weight:

g[m] = g[n] + weight

parents[m] = n

if m in closed\_set:

closed\_set.remove(m)

open\_set.add(m)

if n == None:

print('Path does not exist!')

return None

if n == stop\_node:

path = []

while parents[n] != n:

path.append(n)

n = parents[n]

path.append(start\_node)

path.reverse()

print('Path found: {}'.format(path))

return path

open\_set.remove(n)

closed\_set.add(n)

print('Path does not exist!')

return None

def get\_neighbors(v):

if v in Graph\_nodes:

return Graph\_nodes[v]

else:

return None

def heuristic(n):

H\_dist = {

'A': 11,

'B': 6,

'C': 99,

'D': 1,

'E': 7,

'G': 0,

}

return H\_dist[n]

Graph\_nodes = {

'A': [('B', 2), ('E', 3)],

'B': [('A', 2), ('C', 1), ('G', 9)],

'C': [('B', 1)],

'D': [('E', 6), ('G', 1)],

'E': [('A', 3), ('D', 6)],

'G': [('B', 9), ('D', 1)]

}

aStarAlgo('A', 'G')

**# OUTPUT:**

Path found: ['A', 'E', 'D', 'G']

**2:- AO\* ALGORITM**

class Graph:

def \_\_init\_\_(self, graph, heuristicNodeList, startNode):

self.graph = graph

self.H=heuristicNodeList

self.start=startNode

self.parent={}

self.status={}

self.solutionGraph={}

def applyAOStar(self):

self.aoStar(self.start, False)

def getNeighbors(self, v):

return self.graph.get(v,'')

def getStatus(self,v):

return self.status.get(v,0)

def setStatus(self,v, val):

self.status[v]=val

def getHeuristicNodeValue(self, n):

return self.H.get(n,0)

def setHeuristicNodeValue(self, n, value):

self.H[n]=value

def printSolution(self):

print("FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START NODE:",self.start)

print("------------------------------------------------------------")

print(self.solutionGraph)

print("------------------------------------------------------------")

def computeMinimumCostChildNodes(self, v):

minimumCost=0

costToChildNodeListDict={}

costToChildNodeListDict[minimumCost]=[]

flag=True

for nodeInfoTupleList in self.getNeighbors(v):

cost=0

nodeList=[]

for c, weight in nodeInfoTupleList:

cost=cost+self.getHeuristicNodeValue(c)+weight

nodeList.append(c)

if flag==True:

minimumCost=cost

costToChildNodeListDict[minimumCost]=nodeList

flag=False

else:

if minimumCost>cost:

minimumCost=cost

costToChildNodeListDict[minimumCost]=nodeList

return minimumCost, costToChildNodeListDict[minimumCost]

def aoStar(self, v, backTracking):

print("HEURISTIC VALUES :", self.H)

print("SOLUTION GRAPH :", self.solutionGraph)

print("PROCESSING NODE :", v)

print("-----------------------------------------------------------------------------------------")

if self.getStatus(v) >= 0:

minimumCost, childNodeList = self.computeMinimumCostChildNodes(v)

print(minimumCost, childNodeList)

self.setHeuristicNodeValue(v, minimumCost)

self.setStatus(v,len(childNodeList))

solved=True

for childNode in childNodeList:

self.parent[childNode]=v

if self.getStatus(childNode)!=-1:

solved=solved & False

if solved==True:

self.setStatus(v,-1)

self.solutionGraph[v]=childNodeList

if v!=self.start:

self.aoStar(self.parent[v], True)

if backTracking==False:

for childNode in childNodeList:

self.setStatus(childNode,0)

self.aoStar(childNode, False)

print ("Graph - 2")

h2 = {'A': 1, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}

graph2 = {

'A': [[('B', 1), ('C', 1)], [('D', 1)]],

'B': [[('G', 1)], [('H', 1)]],

'D': [[('E', 1), ('F', 1)]]

}

G2 = Graph(graph2, h2, 'A')

G2.applyAOStar()

G2.printSolution()

**OUTPUT**

Graph - 2

HEURISTIC VALUES : {'A': 1, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}

SOLUTION GRAPH : {}

PROCESSING NODE : A

-----------------------------------------------------------------------------------------

11 ['D']

HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}

SOLUTION GRAPH : {}

PROCESSING NODE : D

-----------------------------------------------------------------------------------------

10 ['E', 'F']

HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}

SOLUTION GRAPH : {}

PROCESSING NODE : A

-----------------------------------------------------------------------------------------

11 ['D']

HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}

SOLUTION GRAPH : {}

PROCESSING NODE : E

-----------------------------------------------------------------------------------------

0 []

HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 10, 'E': 0, 'F': 4, 'G': 5, 'H': 7}

SOLUTION GRAPH : {'E': []}

PROCESSING NODE : D

-----------------------------------------------------------------------------------------

6 ['E', 'F']

HEURISTIC VALUES : {'A': 11, 'B': 6, 'C': 12, 'D': 6, 'E': 0, 'F': 4, 'G': 5, 'H': 7}

SOLUTION GRAPH : {'E': []}

PROCESSING NODE : A

-----------------------------------------------------------------------------------------

7 ['D']

HEURISTIC VALUES : {'A': 7, 'B': 6, 'C': 12, 'D': 6, 'E': 0, 'F': 4, 'G': 5, 'H': 7}

SOLUTION GRAPH : {'E': []}

PROCESSING NODE : F

-----------------------------------------------------------------------------------------

0 []

HEURISTIC VALUES : {'A': 7, 'B': 6, 'C': 12, 'D': 6, 'E': 0, 'F': 0, 'G': 5, 'H': 7}

SOLUTION GRAPH : {'E': [], 'F': []}

PROCESSING NODE : D

-----------------------------------------------------------------------------------------

2 ['E', 'F']

HEURISTIC VALUES : {'A': 7, 'B': 6, 'C': 12, 'D': 2, 'E': 0, 'F': 0, 'G': 5, 'H': 7}

SOLUTION GRAPH : {'E': [], 'F': [], 'D': ['E', 'F']}

PROCESSING NODE : A

-----------------------------------------------------------------------------------------

3 ['D']

FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START NODE: A

------------------------------------------------------------

{'E': [], 'F': [], 'D': ['E', 'F'], 'A': ['D']}

**3:- CANDIDATE ELIMINATION**

import csv

with open("CandidateElimination.csv") as f:

csv\_file = csv.reader(f)

data = list(csv\_file)

s = data[1][:-1]

g = [['?' for i in range(len(s))] for j in range(len(s))]

for i in data:

if i[-1] == "Yes":

for j in range(len(s)):

if i[j] != s[j]:

s[j] = '?'

g[j][j] = '?'

elif i[-1] == "No":

for j in range(len(s)):

if i[j] != s[j]:

g[j][j] = s[j]

else:

g[j][j] = "?"

print("\nSteps of Candidate Elimination Algorithm", data.index(i) + 1)

print(s)

print(g)

gh = []

for i in g:

for j in i:

if j != '?':

gh.append(i)

break

print("\nFinal specific hypothesis:\n", s)

print("\nFinal general hypothesis:\n", gh)

**OUTPUT**

Steps of Candidate Elimination Algorithm 1 ['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same'] [['?', '?', '?',

'?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?',

'?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Steps of Candidate Elimination Algorithm 2 ['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same'] [['?', '?', '?',

'?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?',

'?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Steps of Candidate Elimination Algorithm 3 ['Sunny', 'Warm', '?', 'Strong', 'Warm', 'Same'] [['?', '?', '?', '?',

'?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?',

'?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Steps of Candidate Elimination Algorithm 4 ['Sunny', 'Warm', '?', 'Strong', 'Warm', 'Same'] [['Sunny', '?', '?', '?',

'?', '?'], ['?', 'Warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?',

'?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', 'Same']]

Steps of Candidate Elimination Algorithm 5 ['Sunny', 'Warm', '?', 'Strong', '?', '?'] [['Sunny', '?', '?', '?', '?',

'?'], ['?', 'Warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?',

'?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Final specific hypothesis:

['Sunny', 'Warm', '?', 'Strong', '?', '?']

Final general hypothesis:

[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?']]

**4:- ID3**

from pprint import pprint

import pandas as pd

df\_tennis = pd.read\_csv('ID3.csv')

def entropy(probs):

import math

return sum([-prob \* math.log(prob, 2) for prob in probs])

def entropy\_of\_list(a\_list):

from collections import Counter

cnt = Counter(x for x in a\_list)

num\_instances = len(a\_list) \* 1.0

probs = [x / num\_instances for x in cnt.values()]

return entropy(probs)

total\_entropy = entropy\_of\_list(df\_tennis['PlayTennis'])

def information\_gain(df, split\_attribute\_name, target\_attribute\_name, trace=0):

df\_split = df.groupby(split\_attribute\_name)

nobs = len(df.index) \* 1.0

df\_agg\_ent = df\_split.agg({target\_attribute\_name: [entropy\_of\_list, lambda x: len(x) / nobs]})[

target\_attribute\_name]

df\_agg\_ent.columns = ['Entropy', 'PropObservations']

new\_entropy = sum(df\_agg\_ent['Entropy'] \* df\_agg\_ent['PropObservations'])

old\_entropy = entropy\_of\_list(df[target\_attribute\_name])

return old\_entropy - new\_entropy

def id3(df, target\_attribute\_name, attribute\_names, default\_class=None): # Tally target attribute

from collections import Counter

cnt = Counter(x for x in df[target\_attribute\_name])

if len(cnt) == 1:

return next(iter(cnt))

elif df.empty or (not attribute\_names):

return default\_class

else:

default\_class = max(cnt.keys())

gainz = [information\_gain(df, attr, target\_attribute\_name)

for attr in attribute\_names]

index\_of\_max = gainz.index(max(gainz))

best\_attr = attribute\_names[index\_of\_max]

tree = {best\_attr: {}}

remaining\_attribute\_names = [

i for i in attribute\_names if i != best\_attr]

for attr\_val, data\_subset in df.groupby(best\_attr):

subtree = id3(data\_subset, target\_attribute\_name,

remaining\_attribute\_names, default\_class)

tree[best\_attr][attr\_val] = subtree

return tree

attribute\_names = list(df\_tennis.columns)

attribute\_names.remove('PlayTennis')

tree = id3(df\_tennis, 'PlayTennis', attribute\_names)

print("\n\nThe Resultant Decision Tree is :\n")

pprint(tree)

**OUTPUT**

The Resultant Decision Tree is :

{'Outlook': {'overcast': 'yes',

'rain': {'Wind': {'strong': 'no', 'weak': 'yes'}},

'sunny': {'Humidity': {'high': 'no', 'normal': 'yes'}}}}

**5:-BACKPOPOGATION**

import numpy as np

X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)

y = np.array(([92], [86], [89]), dtype=float)

X = X/np.amax(X,axis=0)

def sigmoid (x):

return (1/(1 + np.exp(-x)))

def derivatives\_sigmoid(x):

return x \* (1 - x)

epoch=7000

lr=0.1

inputlayer\_neurons = 2

hiddenlayer\_neurons = 3

output\_neurons = 1

wh=np.random.uniform(size=(inputlayer\_neurons,hiddenlayer\_neurons))

bh=np.random.uniform(size=(1,hiddenlayer\_neurons))

wout=np.random.uniform(size=(hiddenlayer\_neurons,output\_neurons))

bout=np.random.uniform(size=(1,output\_neurons))

for i in range(epoch):

hinp1=np.dot(X,wh)

hinp=hinp1 + bh

hlayer\_act = sigmoid(hinp)

outinp1=np.dot(hlayer\_act,wout)

outinp= outinp1+ bout

output = sigmoid(outinp)

EO = y-output

outgrad = derivatives\_sigmoid(output)

d\_output = EO\* outgrad

EH = d\_output.dot(wout.T)

hiddengrad = derivatives\_sigmoid(hlayer\_act)

d\_hiddenlayer = EH \* hiddengrad

wout += hlayer\_act.T.dot(d\_output) \*lr

bout += np.sum(d\_output, axis=0,keepdims=True) \*lr

wh += X.T.dot(d\_hiddenlayer) \*lr

print("Input: \n" + str(X))

print("Actual Output: \n" + str(y))

print("Predicted Output: \n" ,output)

**OUTPUT:-**

Input:

[[0.66666667 1. ]

[0.33333333 0.55555556]

[1. 0.66666667]]

Actual Output:

[[92.]

[86.]

[89.]]

Predicted Output:

[[0.9999991 ]

[0.99999699]

[0.99999915]]

**6:-NBC{Not tested-lab manual}**

import csv

import random

import math

def loadCsv(filename):

lines = csv.reader(open(filename, "r"))

dataset = list(lines)

for i in range(len(dataset)):

dataset[i] = [float(x) for x in dataset[i]]

return dataset

def splitDataset(dataset, splitRatio):

trainSize = int(len(dataset) \* splitRatio)

trainSet = []

copy = list(dataset)

while len(trainSet) < trainSize:

index = random.randrange(len(copy))

trainSet.append(copy.pop(index))

return [trainSet, copy]

def separateByClass(dataset):

separated = {}

for i in range(len(dataset)):

vector = dataset[i]

if (vector[-1] not in separated):

separated[vector[-1]] = []

separated[vector[-1]].append(vector)

return separated

def mean(numbers):

return sum(numbers)/float(len(numbers))

def stdev(numbers):

avg = mean(numbers)

variance = sum([pow(x-avg,2) for x in numbers])/float(len(numbers)-1)

return math.sqrt(variance)

def summarize(dataset):

summaries = [(mean(attribute), stdev(attribute)) for attribute in zip(\*dataset)]

del summaries[-1]

return summaries

def summarizeByClass(dataset):

separated = separateByClass(dataset)

summaries = {}

for classValue, instances in separated.items():

summaries[classValue] = summarize(instances)

return summaries

def calculateProbability(x, mean, stdev):

exponent = math.exp(-(math.pow(x-mean,2)/(2\*math.pow(stdev,2))))

return (1 / (math.sqrt(2\*math.pi) \* stdev)) \* exponent

def calculateClassProbabilities(summaries, inputVector):

probabilities = {}

for classValue, classSummaries in summaries.items():

probabilities[classValue] = 1

for i in range(len(classSummaries)):

mean, stdev = classSummaries[i]

x = inputVector[i]

probabilities[classValue] \*= calculateProbability(x, mean, stdev)

return probabilities

def predict(summaries, inputVector):

probabilities = calculateClassProbabilities(summaries, inputVector)

bestLabel, bestProb = None, -1

for classValue, probability in probabilities.items():

if bestLabel is None or probability > bestProb:

bestProb = probability

bestLabel = classValue

return bestLabel

def getPredictions(summaries, testSet):

predictions = []

for i in range(len(testSet)):

result = predict(summaries, testSet[i])

predictions.append(result)

return predictions

def getAccuracy(testSet, predictions):

correct = 0

for i in range(len(testSet)):

if testSet[i][-1] == predictions[i]:

correct += 1

return (correct/float(len(testSet))) \* 100.0

def main():

filename = 'naivedata.csv'

splitRatio = 0.67

dataset = loadCsv(filename)

trainingSet, testSet = splitDataset(dataset, splitRatio)

print('Split {0} rows into train={1} and test={2} rows'.format(len(dataset),

len(trainingSet), len(testSet)))

# prepare model

summaries = summarizeByClass(trainingSet)

# test model

predictions = getPredictions(summaries, testSet)

accuracy = getAccuracy(testSet, predictions)

print('Accuracy of the classifier is : {0}%'.format(accuracy))

main()

**OUTPUT**

**7:-KMEANS**

import numpy as np

from sklearn.cluster import KMeans

import matplotlib.pyplot as plt

from sklearn.mixture import GaussianMixture

import pandas as pd

X=pd.read\_csv("kmeansdata.csv")

x1 = X['Distance\_Feature'].values

x2 = X['Speeding\_Feature'].values

X = np.array(list(zip(x1, x2))).reshape(len(x1), 2)

plt.plot()

plt.xlim([0, 100])

plt.ylim([0, 50])

plt.title('Dataset')

plt.scatter(x1, x2)

plt.show()

gmm = GaussianMixture(n\_components=3)

gmm.fit(X)

em\_predictions = gmm.predict(X)

print("\nEM predictions")

print(em\_predictions)

print("mean:\n",gmm.means\_)

print('\n')

print("Covariances\n",gmm.covariances\_)

print(X)

plt.title('Exceptation Maximum')

plt.scatter(X[:,0], X[:,1],c=em\_predictions,s=50)

plt.show()

import matplotlib.pyplot as plt1

kmeans = KMeans(n\_clusters=3)

kmeans.fit(X)

print(kmeans.cluster\_centers\_)

print(kmeans.labels\_)

plt.title('KMEANS')

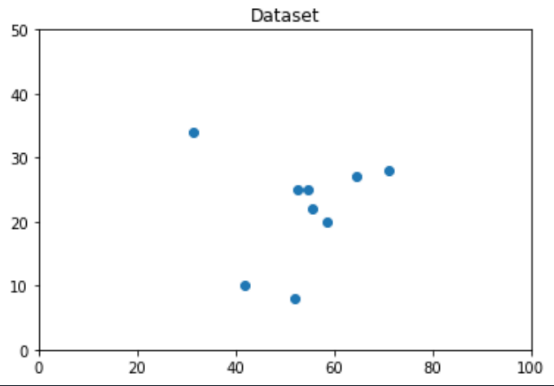
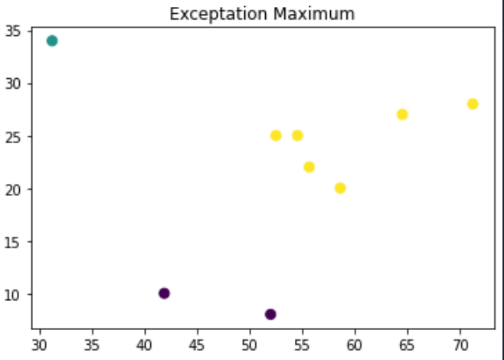
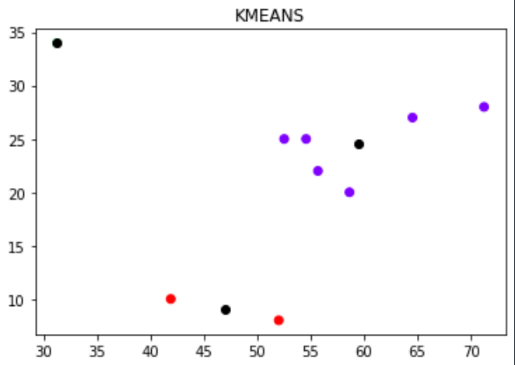
plt1.scatter(X[:,0], X[:,1], c=kmeans.labels\_, cmap='rainbow')

plt1.scatter(kmeans.cluster\_centers\_[:,0] ,kmeans.cluster\_centers\_[:,1], color='black')

**DATASET**

|  |  |  |  |
| --- | --- | --- | --- |
| Driver\_id | Distance\_Feature | Speeding\_Feature | |
| 3.42E+09 | 71.24 | 28 |  |
| 3.42E+09 | 52.53 | 25 |  |
| 3.42E+09 | 64.54 | 27 |  |
| 3.42E+09 | 55.69 | 22 |  |
| 3.42E+09 | 54.58 | 25 |  |
| 3.42E+09 | 41.91 | 10 |  |
| 3.42E+09 | 58.64 | 20 |  |
| 3.42E+09 | 52.02 | 8 |  |
| 3.42E+09 | 31.25 | 34 |  |

**OUTPUT:-**



EM predictions

[2 2 2 2 2 0 2 0 1]

mean:

[[46.965 9. ]

[31.25 34. ]

[59.53666666 24.5 ]]

Covariances

[[[ 2.55530260e+01 -5.05500000e+00]

[-5.05500000e+00 1.00000100e+00]]

[[ 1.00000000e-06 5.55358077e-27]

[ 5.55358077e-27 1.00000000e-06]]

[[ 4.18773566e+01 1.01900001e+01]

[ 1.01900001e+01 7.58333439e+00]]]

[[71.24 28. ]

[52.53 25. ]

[64.54 27. ]

[55.69 22. ]

[54.58 25. ]

[41.91 10. ]

[58.64 20. ]

[52.02 8. ]

[31.25 34. ]]

[[59.53666667 24.5 ]

[31.25 34. ]

[46.965 9. ]]

[0 0 0 0 0 2 0 2 1]

**8:-KNN**

from sklearn.model\_selection import train\_test\_split

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import classification\_report, confusion\_matrix

from sklearn.metrics import accuracy\_score

from sklearn import datasets

iris = datasets.load\_iris()

iris\_data = iris.data

iris\_labels = iris.target

x\_train, x\_test, y\_train, y\_test = train\_test\_split(iris\_data, iris\_labels, test\_size=0.20)

classifier = KNeighborsClassifier(n\_neighbors=5)

classifier.fit(x\_train, y\_train)

y\_pred = classifier.predict(x\_test)

print('Confusion matrix is as follows')

print(confusion\_matrix(y\_test, y\_pred))

print('Accuracy Metrics')

print(classification\_report(y\_test, y\_pred))

print("correct prediction",accuracy\_score(y\_test, y\_pred))

print("wrong prediction",(1-accuracy\_score(y\_test, y\_pred)))

**OUTPUT:-**

Confusion matrix is as follows

[[ 8 0 0]

[ 0 13 0]

[ 0 0 9]]

Accuracy Metrics

precision recall f1-score support

0 1.00 1.00 1.00 8

1 1.00 1.00 1.00 13

2 1.00 1.00 1.00 9

accuracy 1.00 30

macro avg 1.00 1.00 1.00 30

weighted avg 1.00 1.00 1.00 30

correct prediction 1.0

wrong prediction 0.0

**9:- REGRESSION**

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

**def** local\_regression(x0, X, Y, tau):

x0 **=** [1, x0]

X **=** [[1, i] **for** i **in** X]

X **=** np**.**asarray(X)

xw **=** (X**.**T) **\*** np**.**exp(np**.**sum((X **-** x0) **\*\*** 2, axis**=**1) **/** (**-**2 **\*** tau))

beta **=** np**.**linalg**.**pinv(xw **@** X) **@** xw **@** Y **@** x0

**return** beta

**def** draw(tau):

prediction **=** [local\_regression(x0, X, Y, tau) **for** x0 **in** domain]

plt**.**plot(X, Y, 'o', color**=**'black')

plt**.**plot(domain, prediction, color**=**'red')

plt**.**show()

X **=** np**.**linspace(**-**3, 3, num**=**1000)

domain **=** X

Y **=** np**.**log(np**.**abs(X **\*\*** 2 **-** 1) **+** .5)

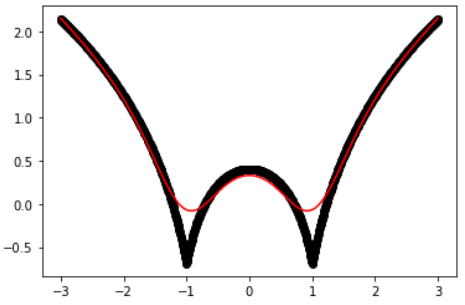
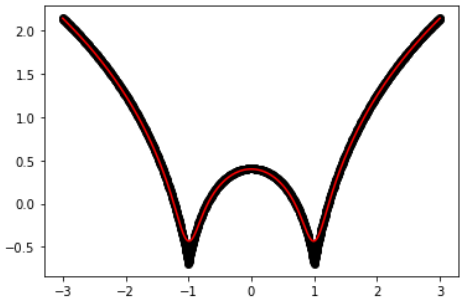
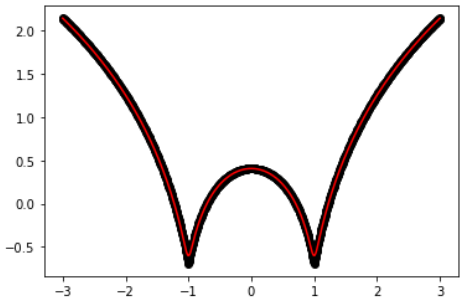
draw(10)

draw(0.1)

draw(0.01)

draw(0.001)

**OUTPUT**

****